Processing Interaction Protocols in Parallel: a Logic Programming implementation for Robotic Soccer

Mariano Tucat [1] Alejandro J. García [2]

Artificial Intelligence Research and Development Laboratory
Department of Computer Science and Engineering
Universidad Nacional del Sur
Av. Alem 1253, (8000) Bahía Blanca, Argentina
e-mail: {mt,ajg}@cs.uns.edu.ar

Abstract

In this paper we analyze different forms of processing in parallel multi-agent interaction protocols. These protocols will be oriented to a robotic soccer domain with autonomous mobile robots. We will also show how to implement them in an extended logic programming framework. We will explore different situations in which the interaction between agents plays an important role, specially *requirements, queries* and *proposals*.

Requirements arise when an agent asks another one to execute a specific action. A query is used when an agent wants to know certain information that another agent may have. Finally, proposals arise when an agent wants to synchronize the execution of an action with another agent.

We will show one way of implementing these different types of interaction in a specific domain, such as the E-League. The E-League is a robot competition in which two teams of four robots play soccer. The robotic soccer has drawn much attention in the area of multi-agent system research and development. This complex and challenging application domain is useful in the evaluation of different kinds of developments that have been carried out in the field of mobile robotic.

Keywords: Agent Communication, Logic Programming, Multi-Agent Systems

1 Introduction

In this paper we analyze different forms of processing in parallel multi-agent interaction protocols. These protocols will be oriented to a robotic soccer domain with autonomous mobile robots. We will also show how to implement them in an extended logic programming framework. We will explore different situations in which the interaction between agents plays an important role, specially *requirements, queries* and *proposals*.

Requirements arise when an agent asks another one to execute a specific action. For example, any agent in the field may ask the agent that controls the robot who has possession of the ball, to pass it to a specific location. Another possible situation in which an agent may use this type of interaction protocol is for example, when it wants to shift its role with another agent, based on specific game situations such as their locations in the field.

[1]Fellowship of Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET).

[2]Partially supported by CONICET (PIP 5050), Universidad Nacional del Sur (24/ZN09), and Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 Nro 13096)

A query is used when an agent wants to know certain information that another agent may have. As an example, an agent may send a message to its teammates, asking them to inform it their location in the field. The agent may also want to know the score of the game, and it may query the agent referee or the agent coach.

A proposal arises when an agent wants to synchronize with another agent for collaboration. In this case, it may propose the execution of the specific action to its teammate and it should wait for the answer. This kind of interaction may be useful, for example, when an agent with possession of the ball wants to pass it to a teammate and it wants to coordinate explicitly this action.

In order to implement the interactions mentioned above, the agents controlling the robots must be able to exchange messages with each other. For this purpose we will use a set of interaction primitives we have developed and reported in [7] (for more details see Appendix B), that were motivated by the implementation of multi-agent systems in dynamic and distributed environments, where intelligent agents communicate and collaborate.

In this work, we will show one way of implementing this different types of interaction in a specific domain: the E-League [2]. The E-League is a robot competition in which two teams of four mobile robots play soccer (for more details see Appendix A), where the challenge is centered on the design of the intelligence of the robots. Considering that each robot can be seem as an agent, this domain offers an ideal situation for the development of a multi-agent system (MAS).

In general, an agent may interact with several agents and these interactions usually proceed simultaneously with the rest of the activities of the agent. Therefore, our proposal is to process these interactions in parallel using the mentioned set of primitives that extends logic programming providing communication among agents.

2 Coordinating a Soccer Team of Mobile Robots

The robotic soccer has drawn much attention in the area of multi-agent system research and development. This complex and challenging application domain is useful in the evaluation of different kinds of developments that have been carried out in the field of mobile robotic. For example, robots playing soccer need to be able to react to the current state of the environment and also try to carry out plans to change the environment to some predicted future state.

One advantage of this particular application domain is that the game of soccer can be seen as a well defined system: the number and type of players, duration of play, allowed behaviors, and penalizations (among other aspects of the game) are governed by a well defined set of rules. Each team is composed of players that should cooperate and also coordinate in order to reach their goal of winning the game.

Coordinating agents [3, 4, 9, 11] in a dynamic environment, such as a mobile robots, is a difficult task. Every agent must be able to contribute to the overall purpose of the multi-agent system efficiently and effectively. The coordination should allow each agent to consider the objectives of the MAS, maintaining its own goals in order to ensure a correct behavior.

As a previous work, we were involved in the developed of a multi-agent system to control a team of robots (called Matebots) for the E-League competition in RoboCup 2004 (see [10]

and Appendix A for details). As explained in [6], the implementation of the MAS was carried out following a layered design. The proposed design was based on services that were associated with four main layers (see Figure 1), each of which covers different levels of abstraction of the problem to be solved, providing services to the upper layers. The implementation of some services could be modified without provoking many changes in other layers using these services.

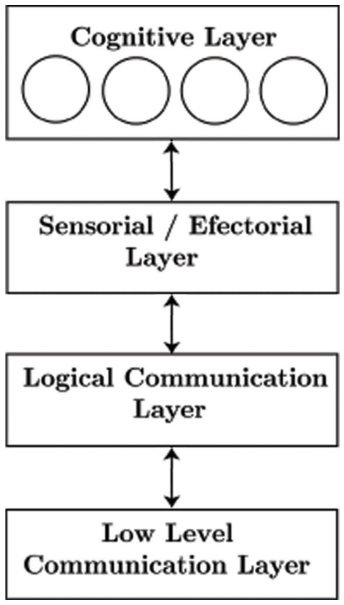

Figure 1: The architecture proposed in [6].

In this work we will explain one way of implementing different interaction protocols that agents with the architecture shown above may use for coordination. This domain of application will allow us to show how the agent may handle the asynchronous arrival of messages from different interaction protocols. In the rest of this section we will briefly explain some details of the mentioned architecture and which part of it will be modified in order to allow the agents to implement interaction protocols.

The Low Level Communication Layer provides access to the basic hardware and software of the league. This includes physical support, such as infrared transmitters, video camera, communication network, and common software.

The Logical Communication Layer offers the necessary services that allow communication with the vision and communication servers are provided. In order to provide communication between the agents, we must extend this layer to include a set of primitives (for more details see Appendix B) that allow them to exchange messages. Having this primitives, an agent may interact with any other agent using some specific protocol.

In the Sensorial/Effectorial Layer, the visual information is processed and translated into terms that express states of the world. The coordinates and speeds of the robots and ball can be interpreted to express particular situations. Query and action predicates related to particular game situations are defined.

The Cognitive Layer is responsible for the design of the agents that implement the team. Like in any other agent system, the agents perceived the environment and reason about the actions to take.

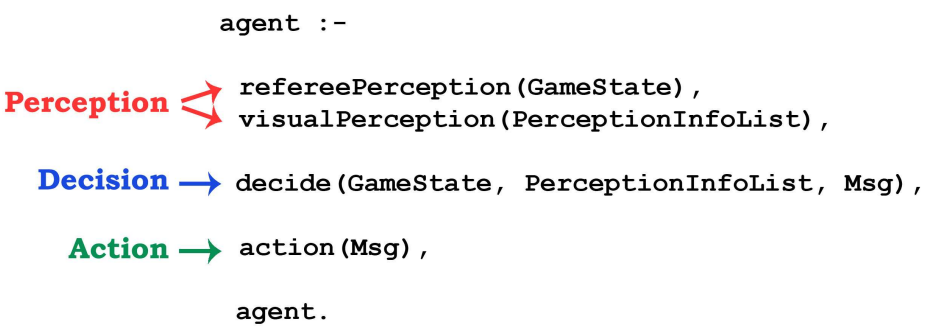

Figure 2: An outline of an agent.

Each agent perceives the state of the game through the referee's decisions, obtains the information of the field of play through the video server and, according to this data, chooses the actions to perform and, finally, the corresponding order is sent to the robot. Figure 2 shows an outline of an agent implemented using logic programming (Prolog). This perception/action loop is what we will modify in order to allow the agents to process the different interaction protocols in parallel with their activities.

3 Processing Interaction Protocols

As stated above, we will explore different situations in which the interaction between agents is important for coordination, in particular those that can be seen as requirements, queries, and proposals. In order to implement these interactions the agents will use the primitives provided by the Logical Communication Layer with the modifications described in the previous section. Thus, an agent may initiate any of the interaction protocol (e.g., request, propose, query) sending the corresponding message to the correct agent and then, it will follow the protocol consequently.

An agent may also receive at any time a message from any other agent, in order to start a conversation, and it should be able to handle this interaction protocol as soon as possible. Its important to note that the messages are received in an asynchronous way, concurrently with the perception/action loop in which the agent decide it actions, and the Logical Communication Layer automatically queued these messages.

There are several alternatives for handling these interaction protocols. One alternative corresponds to modify the perception/action loop (Figure 2) for processing all the messages received from other agents. This can be done by adding a list of received messages to the decision predicate. With this information, the decision predicate can handle every received messages in its decision loop and it can act in consequence.

Although this alternative is simple, it has some disadvantages. The agent perception/action loop must process every received message, and this overhead impose to this cycle delays the agent response to the environment. Also note that this messages may probably be received while the perception/action loop is processing another message or deciding what action the robot should execute. Thus, this messages will not be processed immediately, and this means that the reply (when needed) will also be delayed.

A better alternative can be obtained if the interaction between agents are separated in two types:

(a) When the interaction requires the execution of an action by the agent receiving the message.

(b) When the interaction does not require the execution of an action, such as information queries or some specific information updates.

Observe that, in type (a) the agent should handle this kind of interaction inside its perception/action loop, with a particular priority in its reasoning. While in type (b), these interactions could be considered concurrently to the decision cycle of the agent.

In order to do this, every agent may have a main thread executing the perception/action loop and another thread processing the incoming messages. In this processing message thread, those messages of type (b) will be answered immediately whereas the messages of type (a) will be queued in order to be processed during the decision cycle.

Next, we will show how to implement this last alternative without explicitly programming the two threads of execution. In order to do this, we will use a primitive called `bind_all` (see appendix B), included in the Logical Communication Layer, that allows the association of incoming messages with the execution of a specified predicate. Therefore, the agent will bind the arrival of messages to the execution of a specific predicate for processing them. Thus, this predicate may answer immediately those messages that do not require the execution of an action, without interrupting the agent perception/accion loop or any other active conversation.

4 Implementation Issues

In this section we will show in detail how to implement queries, requirements, and proposals following the standard conversation protocols proposed by FIPA.

First, in Section 4.1, we will show how an agent may ask another one for some specific information. Since this interaction does not require the execution of an action by the agent participant of this interaction, then, it should not interrupt its perception/action loop. The FIPA interaction protocol that will be used is the FIPA Query Protocol.

In Section 4.2 we will show how an agent may require another one for doing some specific action. In this case, the interaction may require the execution of an specific action. Thus, this protocol should be handled in the perception/action cycle of this agent. The FIPA interaction protocol more suitable for this case is the FIPA Request Protocol.

Finally, in Section 4.3, we will show a situation in which an agent wants to synchronize and coordinate the execution of an action with a teammate. In this case, the agent may propose to its party the execution of the specific action. As in the previos situation, this interaction may requiere the execution of an action by the participant, and thus, it should be processed in the decision loop. This kinds of interaction will be modelled by the FIPA Propose Protocol.

4.1 Querying for information

An agent may need, in its decision cycle, the information of the location of another robot in the field. Therefore, it may initiate the FIPA Query Protocol with the agent controlling that robot, and then, it should wait for the answer. Is important to note that the participating agent may not answer immediately, or even, it may not answer at all. Thus, the agent initiating the protocol should consider this alternative in order not to remain blocked waiting for a long time. For this, the agent may use the primitive `receive/4` included in the Logical Communication Layer that allows it to wait for a message for an specified amount of time.

In order to exchange FIPA messages, the agent must send specific information such as the performative, ontology, name of the protocol being used, among some other information. The primitives provided by the Logical Communication Layer allow an agent to send any Prolog term. Thus, one option is to send a list with terms having as functor the parameter name or the word 'performative' and as parameters the parameter value or the performative, respectively. In the case of the message initiating this interaction, the list should be:

```
[ performative(query),sender(Me),receiver(teammate1),protocol(fipa_query),
  ontology(robotic_soccer),content(location(teammate1,X,Y)) ]
```

In the case of the participant of this interaction, it may receive the query at any moment of its decision cycle. Since this query does not require an execution of an action by the robot, the agent may answer it concurrently while decides its next action. Thus, the agent may associate a predicate to the arrival of messages of any other agents using `bind_all/2` (See the first line of Figure 3). When a message is received, the predicate `processMessages/2` is executed automatically. This predicate checks whether the message is a query, asking for the robot location and, in this case, it answers immediately using the internal state of the agent.

```
:- bind_all(processMessages,_).
   % all incoming messages will be processed by processMessages/2

processMessages(_,Message):-
  member(Message,protocol(fipa_query)),       % if the protocol is FIPA Query,
  member(Message,performative(query)),        % the performative is query
  member(Message,content(location(teammate1,X,Y))),% and the ontology is correct,
  member(Message,sender(Sender)),             % obtains the sender of the msg
  my_pos(X,Y),                                % and the information needed
  my_name(Me),
  send(Sender,[sender(Me), receiver(Sender), performative(inform),
          protocol(fipa_query), ontology(robotic_soccer),
          content(location(Me,X,Y))]).        % and, finally, sends the answer
```

Figure 3: An example showing how to handle incoming messages of a particular query.

4.2 Requiring the execution of an action

Another possible interaction between agents are those in which an agent ask another one for doing some specific action, such as passing the ball to it. Contrary to what happened in

the previous situation, this interaction may require the execution of an specific action by the participant of the interaction. Thus, this protocol should be handled in the perception/action cycle of this agent. The FIPA interaction protocol suitable for this case is the FIPA Request Protocol.

Any agent may detect a possible pass to him when another teammate has the ball. Thus, it may send a message asking him to pass the ball and then, it should wait for the answer. An example of a possible message initiating this kind of interaction may be:

```
[ performative(request),sender(Me),receiver(teammate1),protocol(fipa_request),
  ontology(robotic_soccer),content(pass_ball(Me,X,Y)) ]
```

The receiver may not answer immediately and thus, the initiator should be aware of this situation.

The participant of this kind of interaction may receive this request at any moment of its decision cycle. As mentioned before, this protocol should be handled in the perception/action cycle of the agent. One way to do this is having the messages belonging to this kind of interactions queued. The predicate that process the messages, shown in Figure 3, may have one or more rules for each specific interaction protocol managed, specially those that do not require the execution of an action by the robot. Thus, in order to queue the messages that does not belong to one of this interaction protocols, in Prolog, we need only to add a last rule, as shown in Figure 4.

```
processMessages(_,Message):-
    retract(incomingMessages(IM)),
    assert(incomingMessages([Message|IM])).
```

Figure 4: This rule adds the message to the `incomingMessages` queue.

Figure 5 shows how to modify the perception/action loop in order to consider the messages queued as part of it perception information. Observe that the loop obtains the queued messages (using retract/1) and then, empties this queue (see the third and four lines of Perceptions in Figure 5).

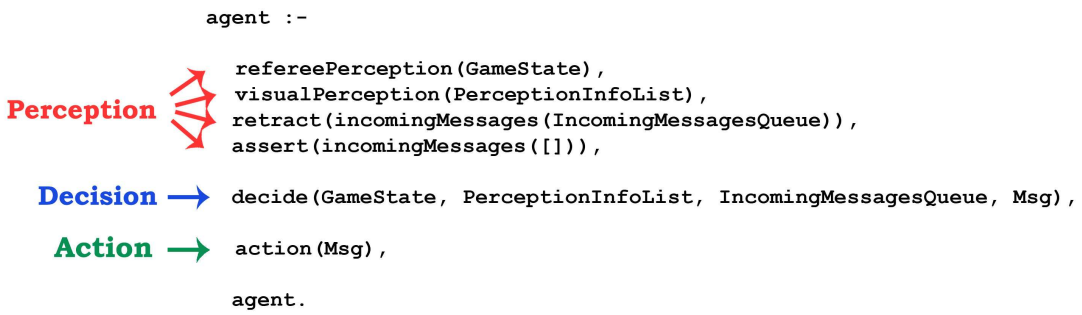

Figure 5: The perception/action loop considering the messages queued.

4.3 Synchronizing the execution of an action

There exists situations in which an agent may want to synchronize and coordinate the execution of an action with a teammate. In this case, the agent may propose the execution of the specific action. As in the previous situation, this interaction may requiere the execution of an action by the participant, and thus, it should be processed in the decision loop. This kinds of interaction can be modelled by the FIPA Propose Protocol.

As an example, the agent with possession of the ball may offer to pass it to a teammate. This agent may decide the execution of this action in its percepction/action loop, and thus, it should send it a message proposing the synchronization of the action. An example of a message initiating this kind of interaction may be:

```
[ performative(propose),sender(Me),receiver(teammate1),protocol(fipa_propose),
  ontology(robotic_soccer),content(pass_ball(teammate1,X,Y)) ]
```

As in the previous situations, the agent initiating the interaction should avoid the possibility of being blocked waiting for the answer.

In the case of the participant of the interaction, the agent receiving this proposal, it should process the offer in it decision cycle, since it may require the execution of an action. This processing may be done in the same way as explained in Figure 4 and Figure 5 in the requirement explained before.

5 Conclusion

In this paper we have analyzed different forms of processing in parallel agent interaction protocols for a robotic soccer domain. We have explored different situations in which the interaction between agents plays an important role, specially those that can be seen as requirements, queries, and proposals. As a design choice, our implementation try to minimize the communication between the agents. Our proposal was to process these interactions in parallel using a set of primitives that extends logic programming providing communication among agents.

Appendix A: E-League

In this appendix we will include some details of the E-League, which is a robot competition in which two teams of four robots play soccer. The league provides common basic services to all of the participants, such as vision and communication. Teams can use low cost kits such as Lego Mindstorms[1] and concentrate on the development and study of Artificial Intelligence techniques such as planning, multi-agent development, communication, and learning. The league's most important feature is its simple and modular structure. There are only three basic components that must be available to obtain a functional team:

- A vision module that works as the robot's perception component,
- A communication module that allows actions to be communicated to the robots,
- And a control module that is implemented by agents that control the robots on the field of play.

Each team has one or more auxiliary computers in which the agents are executed. These agents communicate with the vision component in order to obtain information about what happens on the field, and send messages to the robots by means of a communication module (see Figure 6). Even though the league does not define a standard platform for the construction of the robots, it does impose restrictions over the processing and memory capacity. This allows the use of low cost robotic kits, many of which fall under these restrictions. The system we developed was implemented using Lego Mindstorms kits, which are within the rules of the league.

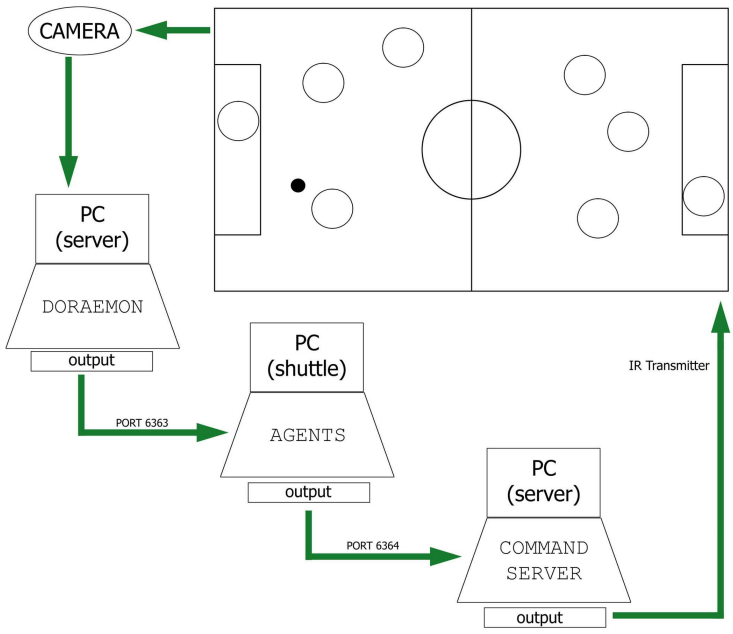

Figure 6: E-League setup (taken from [6])

Each team is composed of four robots. One of the robots can act as goalkeeper, but this is not strictly necessary. There are restrictions over the size of the robots, their shape, and the components used in their construction.

Appendix B: Interaction Primitives

In this appendix we will explain some details of the set of primitives we have developed and reported in [7]. These interaction primitives were motivated by the implementation of multi-agent systems for dynamic and distributed environments, where intelligent agents communicate and collaborate. These primitives provide a transparent way for programming agent interaction; this can be done, for example, by using the agents' logical names without considering low level elements like the actual location of an agent, IP addresses or machine names.

The primitives allow the implementation of standard Agent Communication Languages like FIPA ACL [5] and KQML [8], and provide tools for developing standard Agent Conversation Protocols. Figure 7 shows the list of the implemented primitives.

Primitive:	Type:	Brief description:
connect	setup	Agent initialization and connection to a generic MAS
connect_at	setup	Connection to a particular MAS
disconnect	setup	Deletes the agent from the MAS of which it is a member
my_name	setup	Returns the agent's public name
which_agents	setup	Returns the rest of the participants in the MAS
send	msg.	Sends a message to one or more agents
receive/3	msg.	Waits for a message from another agent
receive/4	msg.	Waits for a message for a given period of time
bind	events	Binds the messages of a specific agent to the call of a predicate
bind_all	events	Binds the messages of any agent to the call of a predicate
unbind	events	Unbind the messages of a specific agent
unbind_all	events	Unbind the messages of any agent
remote_run	exec.	Initiates the execution of a predicate in another connected agent
remote_call	exec.	Consults any other connected agent's knowledge base

Figure 7: Brief description of the set of implemented primitives for agent interaction

The resulting framework has the following features:

1. The implemented primitives allow the creation of several independent multi-agent systems inside a LAN.
2. An agent that joins a MAS can communicate with the other participants by just knowing their names, regardless of which machine they are actually in.
3. Once an agent runs the initialization predicate (`connect`) it obtains a list of the agents present in the system, and thereafter it may send/receive messages in a very simple manner.
4. The basic communication primitives (`send`/`receive`) allow the exchange of Prolog terms.
5. There are primitives (`bind` or `bind_all`) for associating the arrival of a message with the automatic execution of a Prolog predicate, thus allowing event-based programming.
6. Different associations can be made for different agents.
7. An agent can request the remote execution (`remote_run`) of a particular predicate from another agent.
8. An agent can consult another agent's knowledge base (`remote_call`) and use backtracking in order to obtain multiple answers.

References

[1] Lego Mindstorms robots and RCX controllers. http://www.legomindstorms.com.

[2] Official E-League webpage. http://agents.cs.columbia.edu/eleague/.

[3] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[4] T. Finin and Y. Labrou. Agent Communication Languages. In *Proceedings of ASA/MA'99, First International Symposium on Agent Systems and Applications, and Third International Symposium on Mobile Agents*, 1999.

[5] FIPA. Foundation for intelligent physical agents. http://www.fipa.org.

[6] Alejandro J. García, Gerardo I. Simari, and Telma Delladio. Designing an agent system for controlling a robotic soccer team. In *Proceedings of X Argentine Congress of Computer Science*, page 227, 2004. http://cs.uns.edu.ar/∼ajg/papers/index.htm.

[7] Alejandro J. García, Mariano Tucat, and Guillermo R. Simari. Interaction Primitives for Implementing Multi-agent Systems. In *VII Argentine Symposium on Artificial Intelligence*, Rosario, Argentina, August 2005.

[8] KQML. Knowledge Query and Manipulation Language. Oficial Web Page: http://www.cs.umbc.edu/kse/kqml.

[9] Y. Labrou. Standardizing agent communication. In *Proceedings of the Advanced Course on Artificial Intelligence (ACAI'01)*. Springer-Verlag, 2001.

[10] RoboCup. Lisbon, Portugal. 2004. http://www.robocup2004.pt.

[11] M. Wooldridge. Semantic issues in the verification of agent communication languages. In *Journal of Autonomous Agents and Multi Agent Systems*, 2000.